APPENDIX A

network monitor/defender

```
//
// Has two operating modes: if MONITOR is defined, it monitors the network
// instead of defending against DDoS attacks.
//
// ICMP_RATE specifies how many ICMP packets allowed per second. Default is
// 500. UDP_NF_RATE specifies how many non-fragmented UDP (and other non-TCP
// non-ICMP) packets allowed per second. Default is 3000. UDP_F_RATE specifies
// how many fragmented UDP (and other non-TCP non-ICMP) packets allowed per
// second. Default is 1000. All the SNIFF rates specify how many bad packets
// sniffed per second.
//
// For example, if MONITOR is not defiend, and all SNIFF rates are 0, then the
// configuration defends against DDoS attacks, but does not report bad
// packets.
//
// can read:
//   - tcp_monitor: aggregate rates of different TCP packets
//   - ntcp_monitor: aggregate rates of different non TCP packets
//   - icmp_unreach_counter: rate of ICMP unreachable pkts
//   - tcp_ratemon: incoming and outgoing TCP rates, grouped by non-local hosts
//   - ntcp_ratemon: incoming UDP rates, grouped by non-local hosts
//
// Note: handles full fast ethernet, around 134,500 64 byte packets, from
// attacker.
//
//
// TODO:
//   - fragmented packet monitor

#ifndef ICMP_RATE
#define ICMP_RATE         500
#endif

#ifndef UDP_NF_RATE
#define UDP_NF_RATE       2000
#endif

#ifndef UDP_F_RATE
#define UDP_F_RATE        1000
#endif

#ifndef SUSP_SNIFF
#define SUSP_SNIFF        100     // # of suspicious pkts sniffed per sec
```

```
#endif

#ifndef TCP_SNIFF
#define TCP_SNIFF 100      // # of TCP flood pkts sniffed per sec
#endif

#ifndef ICMP_SNIFF
#define ICMP_SNIFF          75       // # of ICMP flood pkts sniffed per sec
#endif

#ifndef UDP_NF_SNIFF
#define UDP_NF_SNIFF        75       // # of non-frag UDP flood pkts sniffed per sec
#endif

#ifndef UDP_F_SNIFF
#define UDP_F_SNIFF         75       // # of frag UDP flood pkts sniffed per sec
#endif

#include "if.click"

#include "sampler.click"

#include "sniffer.click"
ds_sniffer :: Sniffer(mazu_ds);
syn_sniffer :: Sniffer(mazu_syn);
tcp_sniffer :: Sniffer(mazu_tcp);
ntcp_sniffer :: Sniffer(mazu_ntcp);

#include "synkill.click"
#ifdef MONITOR
tcpsynkill :: SYNKill(true);
#else
tcpsynkill :: SYNKill(false);
#endif


//
// discards suspicious packets
//

#include "ds.click"
ds :: DetectSuspicious(01);

from_world -> ds;
ds [0] -> is_tcp_to_victim :: IPClassifier(tcp, -);
```

```
#ifdef MONITOR
ds [1] -> ds_split :: RatedSampler(SUSP_SNIFF);
#else
ds [1] -> ds_split :: RatedSplitter(SUSP_SNIFF);
#endif

ds_split [1] -> ds_sniffer;
ds_split [0]
#ifdef MONITOR
  -> is_tcp_to_victim;
#else
  -> Discard;
#endif

//
// monitor TCP ratio
//

#include "monitor.click"
tcp_ratemon :: TCPTrafficMonitor;

is_tcp_to_victim [0] -> tcp_monitor :: TCPMonitor -> [0] tcp_ratemon;
from_victim -> is_tcp_to_world :: IPClassifier(tcp, -);
is_tcp_to_world [0] -> [1] tcp_ratemon;

//
// enforce correct TCP ratio
//

check_tcp_ratio :: RatioShaper(1,2,40,0.2);
tcp_ratemon [0] -> check_tcp_ratio;

#ifdef MONITOR
check_tcp_ratio [1] -> tcp_split :: RatedSampler(TCP_SNIFF);
#else
check_tcp_ratio [1] -> tcp_split :: RatedSplitter(TCP_SNIFF);
#endif

tcp_split [1] -> tcp_sniffer;
tcp_split [0]
#ifdef MONITOR
  -> [0] tcpsynkill;
#else
  -> Discard;
#endif
```

```
//
// prevent SYN bomb
//

check_tcp_ratio [0] -> [0] tcpsynkill;
tcp_ratemon [1] -> [1] tcpsynkill;

tcpsynkill [0] -> to_victim_s1;
tcpsynkill [1] -> to_world;

tcpsynkill [2]
#ifdef MONITOR
  -> syn_sniffer;
Idle -> to_victim_prio;
#else
  -> tcpsynkill_split :: Tee(2)
tcpsynkill_split [0] -> to_victim_prio;
tcpsynkill_split [1] -> syn_sniffer;
#endif

//
// monitor all non TCP traffic
//

ntcp_ratemon :: IPRateMonitor(PACKETS, 0, 1, 100, 4096, false);
is_tcp_to_victim [1] -> ntcp_monitor :: NonTCPMonitor -> ntcp_t :: Tee(2);
ntcp_t [0] -> [0] ntcp_ratemon [0] -> Discard;
ntcp_t [1] -> [1] ntcp_ratemon;

//
// rate limit ICMP traffic
//

ntcp_ratemon [1] -> is_icmp :: IPClassifier(icmp, -);
is_icmp [0] -> icmp_split :: RatedSplitter (ICMP_RATE);

icmp_split [1] -> to_victim_s2;
icmp_split [0] -> icmp_sample :: RatedSampler (ICMP_SNIFF);

icmp_sample [1] -> ntcp_sniffer;
icmp_sample [0]
#ifdef MONITOR
  -> to_victim_s2;
#else
  -> Discard;
#endif
```

```
//
// rate limit other non TCP traffic (mostly UDP)
//

is_icmp [1] -> is_frag :: Classifier(6/0000, -);

is_frag [0] -> udp_split :: RatedSplitter (UDP_NF_RATE);

udp_split [0] -> udp_sample :: RatedSampler (UDP_NF_SNIFF);
udp_sample [1] -> ntcp_sniffer;
udp_sample [0]
#ifdef MONITOR
  -> to_victim_s2;
#else
  -> Discard;
#endif

is_frag [1] -> udp_f_split :: RatedSplitter (UDP_F_RATE);

udp_f_split [0] -> udp_f_sample :: RatedSampler (UDP_F_SNIFF);
udp_f_sample [1] -> ntcp_sniffer;
udp_f_sample [0]
#ifdef MONITOR
  -> to_victim_s2;
#else
  -> Discard;
#endif

//
// further shape non-TCP traffic with ICMP dest unreachable packets
//

is_tcp_to_world [1] -> is_icmp_unreach :: IPClassifier(icmp type 3, -);
is_icmp_unreach [1] -> to_world;
is_icmp_unreach [0]
  -> icmp_unreach_counter :: Counter;

#ifndef MONITOR

icmp_unreach_counter -> icmperr_sample :: RatedSampler (UNREACH_SNIFF);
icmperr_sample [1] -> ntcp_sniffer;
icmperr_catcher :: AdaptiveShaper(.1, 50);
udp_split [1] -> [0] icmperr_catcher [0] -> to_victim_s2;
udp_f_split [1] -> [0] icmperr_catcher;
icmperr_sample [0] -> [1] icmperr_catcher [1] -> to_world;
```

```
#else

udp_split [1] -> to_victim_s2;
udp_f_split [1] -> to_victim_s2;
icmp_unreach_counter [0] -> to_world;

#endif
```

== if.click

```
//
// input/output ethernet interface for router
//
// this configuration file leaves the following elements to be hooked up:
//
// from_victim:    packets coming from victim
// from_world:     packets coming from world
// to_world:       packets going to world
// to_victim_prio: high priority packets going to victim
// to_victim_s1:   best effort packets going to victim, tickets = 4
// to_victim_s2:   best effort packets going to victim, tickets = 1
//
// see bridge.click for a simple example of how to use this configuration.

// victim network is 1.0.0.0/8 (eth1, 00:C0:95:E2:A8:A0)
//  world network is 2.0.0.0/8 (eth2, 00:C0:95:E2:A8:A1) and
//                   3.0.0.0/8 (eth3, 00:C0:95:E1:B5:38)

// ethernet input/output, forwarding, and arp machinery

tol :: ToLinux;
t :: Tee(6);
t[5] -> tol;

arpq1_prio :: ARPQuerier(1.0.0.1, 00:C0:95:E2:A8:A0);
arpq1_s1 :: ARPQuerier(1.0.0.1, 00:C0:95:E2:A8:A0);
arpq1_s2 :: ARPQuerier(1.0.0.1, 00:C0:95:E2:A8:A0);
ar1 :: ARPResponder(1.0.0.1/32 00:C0:95:E2:A8:A0);
arpq2 :: ARPQuerier(2.0.0.1, 00:C0:95:E2:A8:A1);
ar2 :: ARPResponder(2.0.0.1/32 00:C0:95:E2:A8:A1);
arpq3 :: ARPQuerier(3.0.0.1, 00:C0:95:E1:B5:38);
ar3 :: ARPResponder(3.0.0.1/32 00:C0:95:E1:B5:38);
```

```
psched :: PrioSched;
ssched :: StrideSched (4,1);

out1_s1 :: Queue(256) -> [0] ssched;
out1_s2 :: Queue(256) -> [1] ssched;
out1_prio :: Queue(256) -> [0] psched;
ssched -> [1] psched;
psched[0] -> to_victim_counter :: Counter -> todev1 :: ToDevice(eth1);

out2 :: Queue(1024) -> todev2 :: ToDevice(eth2);
out3 :: Queue(1024) -> todev3 :: ToDevice(eth3);

to_victim_prio :: Counter -> tvpc :: Classifier(16/01, -);
tvpc [0] -> [0]arpq1_prio -> out1_prio;
tvpc [1] -> Discard;

to_victim_s1 :: Counter -> tvs1c :: Classifier(16/01, -);
tvs1c [0] -> [0]arpq1_s1 -> out1_s1;
tvs1c [1] -> Discard;

to_victim_s2 :: Counter -> tvs2c :: Classifier(16/01, -);
tvs2c [0] -> [0]arpq1_s2 -> out1_s2;
tvs2c [1] -> Discard;

to_world :: Counter -> twc :: Classifier(16/02, 16/03, -);
twc [0] -> [0]arpq2 -> out2;
twc [1] -> [0]arpq3 -> out3;
twc [2] -> Discard;

from_victim :: GetIPAddress(16);
from_world :: GetIPAddress(16);

indev1 :: PollDevice(eth1);
c1 :: Classifier (12/0806 20/0001,
            12/0806 20/0002,
                 12/0800,
                 -);
indev1 -> from_victim_counter :: Counter -> c1;
c1 [0] -> ar1 -> out1_s1;
c1 [1] -> t;
c1 [2] -> Strip(14) -> MarkIPHeader -> from_victim;
c1 [3] -> Discard;
t[0] -> [1] arpq1_prio;
t[1] -> [1] arpq1_s1;
t[2] -> [1] arpq1_s2;
```

```
indev2 :: PollDevice(eth2);
c2 :: Classifier (12/0806 20/0001,
            12/0806 20/0002,
                 12/0800,
                 -);
indev2 -> from_attackers_counter :: Counter -> c2;
c2 [0] -> ar2 -> out2;
c2 [1] -> t;
c2 [2] -> Strip(14) -> MarkIPHeader -> from_world;
c2 [3] -> Discard;
t[3] -> [1] arpq2;

indev3 :: PollDevice(eth3);
c3 :: Classifier (12/0806 20/0001,
            12/0806 20/0002,
                 12/0800,
                 -);
indev3 -> c3;
c3 [0] -> ar3 -> out3;
c3 [1] -> t;
c3 [2] -> Strip(14) -> MarkIPHeader -> from_world;
c3 [3] -> Discard;
t[4] -> [1] arpq3;

ScheduleInfo(todev1 10, indev1 1,
         todev2 10, indev2 1,
           todev3 10, indev3 1);
```

== sampler.click

```
elementclass RatedSampler {
 $rate |
  input -> s :: RatedSplitter($rate);
  s [0] -> [0] output;
  s [1] -> t :: Tee;
  t [0] -> [0] output;
  t [1] -> [1] output;
};

elementclass ProbSampler {
 $prob |
  input -> s :: ProbSplitter($prob);
  s [0] -> [0] output;
  s [1] -> t :: Tee;
  t [0] -> [0] output;
  t [1] -> [1] output;
};
```

== sniffer.click

```
// setup a sniffer device, with a testing IP network address
//
// argument: name of the device to setup and send packet to

elementclass Sniffer {
 $dev |
  FromLinux($dev, 192.0.2.0/24) -> Discard;

  input -> sniffer_ctr :: Counter
      -> ToLinuxSniffers($dev);
};

// note: ToLinuxSniffers take 2 us
```

== synkill.click

```
//
// SYNKill
//
// argument: true if monitor only, false if defend
//
// expects: input 0 - TCP packets with IP header to victim network
//          input 1 - TCP packets with IP header to rest of internet
//
// action:  protects against SYN flood by prematurely finishing the three way
//          handshake protocol.
//
// outputs: output 0 - TCP packets to victim network
//          output 1 - TCP packets to rest of internet
//          output 2 - control packets (created by TCPSYNProxy) to victim
//

elementclass SYNKill {
 $monitor |
  // TCPSYNProxy(MAX_CONNS, THRESH, MIN_TIMEOUT, MAX_TIMEOUT,
PASSIVE);
  tcpsynproxy :: TCPSYNProxy(128, 4, 8, 80, $monitor);
  input [0] -> [0] tcpsynproxy [0] -> [0] output;
  input [1] -> [1] tcpsynproxy [1] -> [1] output;
  tcpsynproxy [2]
    -> GetIPAddress(16)
    -> [2] output;
};

== ds.click
```

```
//
// DetectSuspicious
//
// argument: takes in the victim network address and mask. for example:
//    DetectSuspicious(121A0400%FFFFFF00)
//
// expects: IP packets.
//
// action: detects packets with bad source addresses;
//         detects direct broadcast packets;
//         detects ICMP redirects.
//
// outputs: output 0 push out accepted packets, unmodified;
//          output 1 push out rejected packets, unmodified.
//

elementclass DetectSuspicious {
  $vnet |

  // see http://www.ietf.org/internet-drafts/draft-manning-dsua-03.txt for a
  // list of bad source addresses to block out. we also block out packets with
  // broadcast dst addresses.

  bad_addr_filter :: Classifier(
    12/$vnet,               // port  0: victim network address
    12/00,                  // port  1: 0.0.0.0/8 (special purpose)
    12/7F,                  // port  2: 127.0.0.0/8 (loopback)
    12/0A,                  // port  3: 10.0.0.0/8 (private network)
    12/AC10%FFF0,           // port  4: 172.16.0.0/12 (private network)
    12/C0A8,                // port  5: 192.168.0.0/16 (private network)
    12/A9FE,                // port  6: 169.254.0.0/16 (autoconf addr)
    12/C0000200%FFFFFF00,   // port  7: 192.0.2.0/24 (testing addr)
    12/E0%F0,               // port  8: 224.0.0.0/4 (class D - multicast)
    12/F0%F0,               // port  9: 240.0.0.0/4 (class E - reserved)
    12/00FFFFFF%00FFFFFF,   // port 10: broadcast saddr X.255.255.255
```

```
   12/0000FFFF%0000FFFF,     // port 11: broadcast saddr X.Y.255.255
   12/000000FF%000000FF,     // port 12: broadcast saddr X.Y.Z.255
   16/00FFFFFF%00FFFFFF,     // port 13: broadcast daddr X.255.255.255
   16/0000FFFF%0000FFFF,     // port 14: broadcast daddr X.Y.255.255
   16/000000FF%000000FF,     // port 15: broadcast daddr X.Y.Z.255
    9/01,                  // port 16: ICMP packets
   -);

  input -> bad_addr_filter;
  bad_addr_filter [0]  -> [1] output;
  bad_addr_filter [1]  -> [1] output;
  bad_addr_filter [2]  -> [1] output;
  bad_addr_filter [3]  -> [1] output;
  bad_addr_filter [4]  -> [1] output;
  bad_addr_filter [5]  -> [1] output;
  bad_addr_filter [6]  -> [1] output;
  bad_addr_filter [7]  -> [1] output;
  bad_addr_filter [8]  -> [1] output;
  bad_addr_filter [9]  -> [1] output;
  bad_addr_filter [10] -> [1] output;
  bad_addr_filter [11] -> [1] output;
  bad_addr_filter [12] -> [1] output;
  bad_addr_filter [13] -> [1] output;
  bad_addr_filter [14] -> [1] output;
  bad_addr_filter [15] -> [1] output;

  // ICMP rules: drop all fragmented and redirect ICMP packets

  bad_addr_filter [16]
   -> is_icmp_frag_packets :: Classifier(6/0000, -);
  is_icmp_frag_packets [1] -> [1] output;

  is_icmp_frag_packets [0]
   -> is_icmp_redirect :: IPClassifier(icmp type 5, -);
  is_icmp_redirect [0] -> [1] output;

  // finally, allow dynamic filtering of bad src addresses we discovered
  // elsewhere in our script.

  dyn_saddr_filter :: AddrFilter(SRC, 32);
  is_icmp_redirect [1] -> dyn_saddr_filter;
  bad_addr_filter [17] -> dyn_saddr_filter;
  dyn_saddr_filter [0] -> [0] output;
  dyn_saddr_filter [1] -> [1] output;

};
```

= monitor.click

```
//
// TCPTrafficMonitor
//
// expects: input 0 takes TCP packets w IP header for the victim network;
//          input 1 takes TCP packets w IP Header from the victim network.
// action:  monitors packets passing by
// outputs: output 0 - packets for victim network, unmodified;
//          output 1 - packets from victim network, unmodified.
//

elementclass TCPTrafficMonitor {
  // fwd annotation = rate of src_addr, rev annotation = rate of dst_addr
  tcp_rm :: IPRateMonitor(PACKETS, 0, 1, 100, 4096, true);

  // monitor all TCP traffic to victim, monitor non-RST packets from victim
  input [0] -> [0] tcp_rm [0] -> [0] output;
  input [1] -> i1_tcp_rst :: IPClassifier(rst, -);
  i1_tcp_rst[0] -> [1] output;
  i1_tcp_rst[1] -> [1] tcp_rm [1] -> [1] output;
};
```